Chapter 6

C More I/O with `gets()` and `puts()`

In This Chapter

- Reading strings of text with `gets()`
- Avoiding some `gets()` problems
- Using `puts()` to display text
- Displaying variables with `puts()`
- Knowing whether to use `puts()` or `printf()`

The `printf()` and `scanf()` functions aren't the only way you can display information or read text from the keyboard — that old I/O. No, the C language is full of I/O tricks, and when you find out how limited and lame `printf()` and `scanf()` are, you will probably create your own functions that read the keyboard and display information just the way you like. Until then, you're stuck with what C offers.

This chapter introduces the simple `gets()` and `puts()` functions. `gets()` reads a string of text from the keyboard, and `puts()` displays a string of text on the screen.

The More I Want, the More I `gets()`

Compared to `scanf()`, the `gets()` function is nice and simple. Both do the same thing: They read characters from the keyboard and save them in a variable. `gets()` reads in only text, however. `scanf()` can read in numeric values and strings and in a number of combinations. That makes it valuable, but for reading in text, clunky.

Like `scanf()` reading in text, `gets()` requires a `char` variable to store what's entered. It reads everything typed at the keyboard until the Enter key is pressed. Here's the format:

```
gets(var);
```

`gets()`, like all functions, is followed by a set of parentheses. Because `gets()` is a complete statement, it always ends in a semicolon. Inside the parentheses is *var,* the name of the string variable text in which it is stored.

Another completely rude program example

The following is the INSULT1.C program. This program is almost identical to the WHORU.C program, introduced in Chapter 4, except that `gets()` is used rather than `scanf()`.

```
#include <stdio.h>

int main()
{
    char jerk[20];

    printf("Name some jerk you know:");
    gets(jerk);
    printf("Yeah, I think %s is a jerk, too.\n",jerk);
    return(0);
}
```

Enter this source code into your editor. Save the file to disk and name it INSULT1.C.

Compile the program. Reedit the text if you find any errors. Remember your semicolons and watch how the double quotes are used in the `printf()` functions.

Run the resulting program. The output looks something like this:

```
Name some jerk you know:Bill
Yeah, I think Bill is a jerk, too.
```

- `gets()` reads a variable just like `scanf()` does. Yet no matter what reads it, the `printf()` statement can display it.
- `gets(var)` is the same as `scanf("%s",var)`.
- If you get a warning error when compiling, see the next section.

You can pronounce `gets()` as "get-string" in your head. "Get a string of text from the keyboard." However, it probably stands for "Get stdin," which means "Get from standard input." "Get string" works for me, though.

And now, the bad news about `gets()`

The latest news from the C language grapevine is *not* to use the `gets()` function, at least not in any serious, secure programs you plan on writing. That's because `gets()` is not considered a safe, secure function to use.

The reason for this warning — which may even appear when you compile a program using `gets()` — is that you can type more characters at the keyboard than were designed to fit inside the `char` variable associated with `gets()`. This flaw, known as a *keyboard overflow,* is used by many of the bad guys out there to write worms and viruses and otherwise exploit well-meaning programs.

For the duration of this book, don't worry about using `gets()`. It's okay here as a quick way to get input while finding out how to use C. But for "real" programs that you write, I recommend concocting your own keyboard-reading functions.

The Virtues of `puts()`

In a way, the `puts()` function is a simplified version of the `printf()` function. `puts()` displays a string of text, but without all `printf()`'s formatting magic. `puts()` is just a boneheaded "Yup, I display this on the screen" command. Here's the format:

```
puts(text);
```

`puts()` is followed by a left paren, and then comes the *`text`* you want to display. That can either be a string variable name or a string of text in double quotes. That's followed by a right paren. The `puts()` function is a complete C language statement, so it always ends with a semicolon.

The `puts()` function's output always ends with a newline character, `\n`. It's like `puts()` "presses Enter" after displaying the text. You cannot avoid this side effect, though sometimes it does come in handy.

Another silly command-prompt program

To see how `puts()` works, create the following program, STOP.C. Yeah, this program is really silly, but you're just starting out, so bear with me:

```
#include <stdio.h>

int main()
{
    puts("Unable to stop: Bad mood error.");
    return(0);
}
```

Save this source code to disk as STOP.C. Compile it, link it, run it.

This program produces the following output when you type **stop** or **./stop** at the command prompt:

```
Unable to stop: Bad mood error.
```

Ha, ha.

- `puts()` is not pronounced "putz."
- Like `printf()`, `puts()` slaps a string of text up on the screen. The text is hugged by double quotes and is nestled between two parentheses.
- Like `printf()`, `puts()` understands escape sequences. For example, you can use `\"` if you want to display a string with a double quote in it.
- You don't have to put a `\n` at the end of a `puts()` text string. `puts()` always displays the newline character at the end of its output.
- If you want `puts()` not to display the newline character, you must use `printf()` instead.

`puts()` and `gets()` in action

The following program is a subtle modification to INSULT1.C. This time, the first `printf()` is replaced with a `puts()` statement:

```
#include <stdio.h>

int main()
{
    char jerk[20];
```

```
    puts("Name some jerk you know:");
    gets(jerk);
    printf("Yeah, I think %s is a jerk, too.",jerk);
    return(0);
}
```

Load the source code for INSULT1.C into your editor. Change Line 7 so that it reads as just shown; the `printf` is changed to `puts`.

Use your editor's Save As command to give this modified source code a new name on disk: INSULT2.C. Save. Compile. Run.

```
Name some jerk you know:
Rebecca
Yeah, I think Rebecca is a jerk, too.
```

Note that the first string displayed (by `puts()`) has that newline appear afterward. That's why input takes place on the next line. But considering how many command-line or text-based programs do that, it's really no big deal. Otherwise, the program runs the same as INSULT1. But you're not done yet; continue reading with the next section.

More insults

The following source code is another modification to the INSULT series of programs. This time, you replace the final `printf()` with a `puts()` statement. Here's how it looks:

```
#include <stdio.h>

int main()
{
    char jerk[20];

    puts("Name some jerk you know:");
    gets(jerk);
    puts("Yeah, I think %s is a jerk, too.",jerk);
    return(0);
}
```

Load the source code for INSULT2.C into your editor. Make the changes just noted, basically replacing the `printf` in Line 9 with puts. Otherwise, the rest of the code is the same.

Save the new source code to disk as INSULT3.C. Compile and run.

Whoops! Error! Error!

```
Insult3.c:9: too many arguments to function 'puts'
```

The compiler is smart enough to notice that more than one item appears to be specified for the `puts()` function; it sees a string, and then a variable is specified. According to what the compiler knows, you need only one or the other, not both. Oops.

- `puts()` is just not a simpler `printf()`.
- If you got the program to run — and some compilers may — the output looks like this:

```
Name some jerk you know:
Bruce
Yeah, I think that %s is a jerk, too.
```

Ack! Who is this `%s` person who is such a jerk? Who knows! Remember that `puts()` isn't `printf()`, and it does not process variables the same way. To `puts()`, the `%s` in a string is just `%s` — characters — nothing special.

`puts()` *can print variables*

`puts()` can display a string variable, but only on a line by itself. Why a line by itself? Because no matter what, `puts()` always tacks on that pesky newline character. You cannot blend a variable into another string of text by using the `puts()` function.

Consider the following source code, the last in the INSULT line of programs:

```
#include <stdio.h>

int main()
{
    char jerk[20];

    puts("Name some jerk you know:");
    gets(jerk);
    puts("Yeah, I think");
    puts(jerk);
    puts("is a jerk, too.");
    return(0);
}
```

Feel free to make the preceding modifications to your INSULT3.C program in your editor. Save the changes to disk as INSULT4.C. Compile. Run.

```
Name some jerk you know:
David
Yeah, I think
David
is a jerk, too.
```

The output looks funky, like one of those "you may be the first person on your block" sweepstakes junk mailers. But the program works the way it was intended.

- Rather than replace `printf()` with `puts()`, you have to rethink your program's strategy. For one, `puts()` automatically sticks a newline on the end of a string it displays. No more strings ending in `\n`! Second, `puts()` can display only one string variable at a time, all by itself, on its own line. And, last, the next bit of code shows the program the way it should be written by using only `puts()` and `gets()`.

- You must first "declare" a string variable in your program by using the `char` keyword. Then you must stick something in the variable, which you can do by using the `scanf()` or `gets` function. Only then does displaying the variable's contents by using `puts()` make any sense.

- Do not use `puts()` with a nonstring variable. The output is weird. (See Chapter 8 for the lowdown on variables.)

When to use puts()
When to use printf()

- Use `puts()` to display a single line of text — nothing fancy.
- Use `puts()` to display the contents of a string variable on a line by itself.
- Use `printf()` to display the contents of a variable nestled in the middle of another string.
- Use `printf()` to display the contents of more than one variable at a time.
- Use `printf()` when you don't want the newline (Enter) character to be displayed after every line, such as when you're prompting for input.
- Use `printf()` when fancy formatted output is required.